Security Analysis of the Lightning Network

Laolu Osuntokun
@roasbeef
Lightning Labs

BPASE 2017
State of the Hash-Lock

● In-progress Lightning Network specifications ([lightning-rfc](https://www.lightning-rfc.org))
  ○ Basis of Lightning Technology (BOLT)
  ○ Specs cover: funding process, key derivation, p2p interaction, messages, etc.
● 4+ implementations being actively developed on Bitcoin’s Testnet, e.g:
  ○ [lightningd](https://github.com/c-lightning/lightningd) (c-lightning)
  ○ lit
  ○ eclair
  ○ lnd
● Testnet Lightning Faucet
  ○ Opens a channel instead of sending on-chain
  ○ Easy way for application developers to:
    ■ Get testnet coins
    ■ Starting experimenting with Lightning
● More thinking around cross-chain Lightning Networks
Outline

1. Lightning in 2 Slides
2. On-Chain Liveliness Assumptions
3. Peer-to-Peer Networking Layer
4. Onion Encoded Payment Routes (Sphinx)
5. Hash-Lock Decorrelation
6. Blinded Channel Outsourcing
Lighting in 2-Slides (½)

- “Vanilla” Bitcoin workflow:
  - Alice ➜ Bob
    - Bob gives Alice Bitcoin Address
    - Alice broadcasts tx to network
    - Entire network verifies payment
    - Payment confirmed in block
    - Bob gets coinz

- Issues:
  - Each payment requires resources for each full-node
  - Confirmation times unpredictable
  - Global broadcast doesn’t scale.
Lighting in 2-Slides (2/2)

- Enter Lightning: **Off-Chain** Bitcoin payments
- Alice and Bob enter into a **contract**
  - **Contract creation:**
    - Funds put into 2-of-2 multi-sig
    - **Before broadcast** transaction to *deliver* funds is signed
      - Requires malleability fix
    - Funding transaction broadcast
- Off-chain payments (sub-contract):
  - **HTLC**: Hash-Time-Lock-Contract
- **Contract completion:**
  - Closing transaction broadcast, final balance delivered
- **On-chain footprint:**
  - 2 transactions
  - All updates **point-to-point**
  - **Predictable** fees
On-Chain Liveliness

● Security model of Lightning:
  ○ Relies on Bitcoin for **ordering of transactions**
  ○ Dependent on **time-based** windows of action (\(T\))
    ■ Longer \(T\) (CSV delay) provides more security during **channel breaches**
    ■ Longer \(T\) also results in unavailability of funds for **unilateral closes**
    ■ In the optimistic case: higher \(T\), as closures assumed to be **cooperative**

● Thundering herd failure mode
  ○ Massive **network-wide channel closure** clogs up chain
  ○ Depending on \(T\) (unique to each channel), and duration of backlog, adversaries may profit

● Possible solutions:
  ○ Time-stop
    ■ “Relative time-lock” stops ticking above “higher-water” mark
  ○ Consensus enforced transaction **dependency**
    ■ Covenant or op-code
  ○ Fee-based dependency: CPFP
Peer-to-Peer Networking Layer

- **All** communications between nodes **encrypted+authenticated:**
  - No protocol messages sent until **brontide** session initiated
- **Brontide (BOLT #8):**
  - Variant of the **Noise Protocol Framework** (brontide):
    - Framework for **Authenticated Key Agreement**
    - Init: series of **handshake** messages (ECDH+hashing)
    - Transport: **AEAD** cipher mode used for encryption
  - Noise_XK_secp256k1_ChaChaPoly_SHA256
    - \(- s \)
    - \(\rightarrow e, es \)
    - \(<- e, ee \)
    - \(\rightarrow s, se \)
  - Hash ratchet for **key rotation**
Peer-to-Peer Networking Layer

- Nodes identified on the network by **public key**
- **Bitcoin keys** and **node keys** used to authenticate information
  - Node Announcement:
    - Announces node existence: PubKey+sig, reachability
    - **Global features**
  - Channel Announcement (channel proof):
    - Channel ID: **location** of funding output in chain (8-bytes)
    - 4 keys: two multi-sig keys, two node keys
      - Verify: 2 <key1> <key2> 2 OP_CHECKMULTISIG
    - 4 sigs:
      - Can be compressed to single key w/ **signature aggregation**
      - Verification can be sped up via **batch signature** verification
  - Channel Update Announcement:
    - Advertises **routing policy** for a **directed** channel edge
    - Signed by node advertising
Onion Encoded Payment Routes (Sphinx)

- **Sphinx**: compact, provably secure **mix-net** packet format
  - Used within lightning as basis for **onion routing**
  - **Fixed-sized** payload
  - Modified version in **BOLT #4**

- **Security Features**
  - Nodes don’t know their **location** in the route
    - Packet remains **fixed-sized** during processing
  - Nodes don’t know **how long** the route really was
    - All packets encode the **max hop** limit
  - Nodes only know their predecessor and successor
    - Received from downstream node, contains instructions to forward
  - All packets **indistinguishable** from all others
    - Packet is **re-randomized** at each hop

- **Shared secret re-used to back-propagate error messages**
Onion Encoded Payment Routes (Sphinx)

- Payments routed through network using **source routing:**
  - Gives sender **total control** over payment path
  - Authenticated per-hop payload:
    - Outgoing time-lock (#blocks or time)
    - Satoshis to forward (ensure proper fees)
    - Outgoing “realm” (Bitcoin, Litecoin, etc)
- Replay attack prevention:
  - Each Sphinx packet commits to the payment **hash**
  - HTLC’s past **absolute** are rejected
- Routes still subject to **traffic+timing** analysis
  - Poor path diversity weakens security
Hash-Lock Decorrelation

If Bob and Dave wish to collude then they’ve correlated the route.

This correlation mitigates our unlikability
Hash-Lock Decorrelation

- **Problem:**
  - The hash-lock (payment hash) is identical over entire route

- **Solution:**
  - Decorrelate the hash-locks via re-randomization
  - Similar to Sphinx’s “One little trick”

- **Construction (one of many):**
  - Switch from hash-locks to “key-locks” (for path length y)
    - Sender calc’s: $Q_i = Q + R_i + R_{i-1} + \ldots + R_y$ (multi-scalar mult)
  - $Q = G^*q_i$ -> Key-Lock from incoming KTLC (need the private key to settle)
  - $r$ -> Scalar encoded within payload
  - $P = Q + r^*G = G^*q + G^*r = G^*(q+r)$, used for outgoing payment
  - Once $p = (r+q)$ is revealed: calc $p=r-q$

- **Introduces new causal dependency** into the payment contract:
  - Funds must be pulled in the reverse order
Blinded Channel Outsourcing

- Lightning requires parties to occasionally **monitor** the blockchain
  - TEE based schemes (e.g. Teechan) can help
  - Each “pay-to-self” output has a **relative time-lock** (CSV)
  - Delay acts as **adjudication** period

- Responsibility for watching the chain can be outsourced to a third-party
  - With **SIGHASH_NOINPUT** or MAST, can reduce storage to $O(\log(N))$
    - $N = number\_of\_commitment\_updates$
  - For now, we can at least make the process more **private**

- Blinded Channel Monitoring
  - Outsourcer shouldn’t be able to distinguish **which** channel they’re watching
  - Achieved by **randomizing** commitment keys on each update
  - Able to **collapse** the revocation state, saving disk-space
    - “Reverse” merkle-tree -- once you have parent, can **discard** children
Blinded Channel Outsourcing

● Incentivization:
  ○ Can pre-pay outsourcer for their work
  ○ Possibly add a bonus upon adjudication

● Sybil Resistance
  ○ Outsourcer doesn’t know which channel (due to blinding)
  ○ Therefore, client might be feeding garbage data

● Possible solution:
  ○ Linkable ring-signature over some subset of channel graph
  ○ Linkability allows outsourcer to ensure unique service per user
● Think this stuff is interesting?
  ○ **Lightning Labs** hiring
    ■ Crypto Protocol Engineers
    ■ Mobile Engineers
  ○ Email: [laolu@lightning.network](mailto:laolu@lightning.network)
  ○ Twitter: [@roasbeef](https://twitter.com/roasbeef)